(2)

Naval Research Laboratory
Washington, DC 20375-5000

NRL Memorandum Report 6552

AD-A213 136

A Standard Organization for Specifying Abstract Interfaces for the SMMS Application

JEAN T. QUINN, ALAN R. BULL AND ALEXANDRA L. EVANS*

Center for Secure Information Technology
Information Technology Division

**Software A&E*
Arlington, VA 22209

DTIC
S ELECTE D
OCT 0 5 1989
D

September 20, 1989

Approved for public release; distribution unlimited.

89 10 4 051

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Memorandum Report 6552	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b OFFICE SYMBOL *(If applicable)*	7a NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20375-5000	7b ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SPAWAR 8b OFFICE SYMBOL *(If applicable)*	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20360	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
RDTEDA			

11 TITLE *(Include Security Classification)*
A Standard Organization for Specifying Abstract Interfaces for the SMMS Application

12 PERSONAL AUTHOR(S)
Quinn, J.T., Bull, A.R. and Evans,* A.L.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
Interim	FROM 2/89 TO 7/89	1989 September 20	20

16 SUPPLEMENTARY NOTATION
*Software A&E, Arlington, VA 22209

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer security Trusted systems

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

One of the goals of NRL's Secure Military Message System (SMMS) project is to demonstrate the feasibility of applying advanced software engineering techniques to multi-level secure data base systems to provide increased assurance and to simplify maintenance. To demonstrate these principles, software for a multi-level secure, fully-functional military message system is being designed and implemented. The project is producing a set of model procedures and documents that can be followed by designers and producers of other such systems.

This document describes the format to be followed in documenting abstract interfaces of the software modules. An abstract interface is a software module interface that will remain unchanged even when details of the software implementation change; specifying such interfaces is a key to building easy-to-change software systems. The format described in this report is designed to serve the author who designs a module, the coder who implements it, designers of other modules that must make use of it, and reviewers who must

(Continues)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Alan R. Bull	22b TELEPHONE *(Include Area Code)* (202) 767- 22c OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 UNCLASSIFIED

19. ABSTRACT (Continued)

approve its design. The format is also intended to aid reviewers in assessing whether a module satisfies its abstract interface specification, thus meeting its security requirements. It organizes the specifications into a small number of concise, well-defined sections, allowing readers who are searching for a particular kind of information to know where to look.

The format for specifying abstract interfaces documented in this report is an adaptation of that developed by the NRL Software Cost Reduction project, and much of this report derives directly from a similarly titled document (NRL Report 8815) developed for that project.

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

A STANDARD ORGANIZATION FOR SPECIFYING ABSTRACT INTERFACES FOR THE SMMS APPLICATION

1. INTRODUCTION

There are three major tasks in designing a software system. The first is partitioning the system into work assignments (modules). The second is designing the interface of each module, i.e., deciding what facilities the module will provide. The third is producing the specification for each interface so that (a) the implementers have enough information to write the software; (b) writers of other modules have enough information to use the module; (c) information that constrains or discloses details of the implementation is not revealed; (d) reviewers can determine that the specified module behavior is consistent with the module's assigned security requirements.

NRL's Secure Military Message Systems (SMMS) project is applying the NRL Software Cost Reduction (SCR) methodology [SCRSO] to the specification, design, and implementation of a family of military message systems that must meet demanding security requirements. Information-hiding [CRIT] and proof-based decomposition [SAP] are approaches taken in the first two tasks; abstract interface specification [ABS] is the approach taken for the third task.

Information-hiding [CRIT] is a method of designing software to reduce and make more predictable, the impact (and hence, the cost) of making software changes. The method involves dividing the software into modules according to likely changes; each module is responsible for encapsulating or "hiding" the effects of a particular change from the rest of the system. The key is to design the interface of a module so that it reveals only information about the module that is **not** likely to change. In that way, when changes that affect a module are required, only the implementation of that module is required to be modified. The interface and all other modules that use the interface are not likely to change at all.

Such an interface is called an **abstract interface,** because it is an **abstraction** of what makes up a module in the same way that a road map of the world is an abstraction of the world. There are many things about the world that could change (e.g., the location and number of all the buildings, trees, people, etc.), but a user of a road map is not concerned with these things; hence, the map does not necessarily change. The specification of an abstract interface should have the following properties:

- It must not disclose any of the changeable aspects ("secrets") of the module.
- It must present a concise description of the facilities available from the module, in terms of effects that are observable to the user;
- It should be divided into sections and formatted so that a reader unfamiliar with the module is able to find a piece of information without having to study the entire interface specification; i.e., it should serve the quick-reference reader as well as the first-time reader;
- It should not duplicate information, which would make using and maintaining the document more difficult.

The overall organization chosen to achieve these properties appears in Table 1. Complete descriptions of each section can be found in Section 2.

Manuscript approved July 28, 1989.

This organization is somewhat different from the SCR approach, which was geared to software for hard real-time-systems. Such systems typically have been programmed in an assembly language rather than a higher order language and they must often control a variety of concurrently operating sensors and devices with varying deadlines. The SMMS application will be programmed in higher order languages, and it will not generally have to meet hard real time constraints, but it will have to meet stringent security requirements. Because of these differences in applications, some parts of the structure defined in [SCRSO] are unused here, and a few new parts have been added.

Specifically, sections documenting user exceptions and security exceptions have been added to the standard organization. The NRL SCR methodology anticipates that a system could operate even if checking for Undesired Events (UEs) were turned off (perhaps for the sake of efficiency). In order to assure that an SMMS family member enforces its security requirements, it should be impossible for the system to operate if

Introduction	A brief prose overview of the module's facilities to help the reader determine if this is the module he is interested in;
Interface Overview	A table of programs on the module's interface, showing the parameters and parameter types and stating the effects of each;
Local Type Definitions	Definitions of the data types available to users from the module;
Dictionary	Definitions of any specialized terms used throughout the specification;
Undesired Event Dictionary	Definitions of any possible incorrect uses by the software of the module's facilities or hardware errors;
User Exception Dictionary	Definitions of any possible incorrect uses of the module's facilities by a human user that result in other than a security violation;
Security Exception Dictionary	Definitions of any possible incorrect uses of the module's facilities by a human user that can result in a security violation;
System Generation Parameters	A list of those quantitative characteristics of the module that are not bound until just before run-time (e.g., the size of a data structure);
Design Issues	A prose section explaining why certain design decisions were made to aid people who might make future changes to the design;
Implementation Notes	A prose section to capture information that might have come to the designer's attention that would be of use to the implementors;
Assumptions Lists	A prose section documenting the assumptions that the users of the module are allowed to make about it.
Unimplemented Features	Specifications of features not provided in initial versions of the software.
Efficiency Guide	Specifications of the run-time resources consumed by using the facilities on the interface.
Mapping to Requirements	Listing relating individual features to individual requirements.
Facilities Index	A quick look-up reference of all programs and terms defined in the specification ;

Table 1 — Organization overview

the checking for certain kinds of exceptions is disabled. In order to distinguish the members of this class that specifically concern security, the class is subdivided into security and user exceptions. The existence of these classes also makes it easier for readers to distinguish conditions that represent potential security violations from other kinds of errors.

A section documenting the mapping from interface programs to the system requirements and security architecture has been added to help readers assure that each module in the design has a valid reason to be there. Unused from the earlier organization are the accuracy guide, which defined the tolerances on data provided by sensors, and events, which permit specification of programs that do not return until a defined condition holds. Matching value/effect program pairs have been left unabbreviated in the interface overview section, and enumerated types have been removed from the local data types section. That these facilities are unused here reflects only on the application requirements and the current SMMS design approach, not on their general utility. Future revisions to the SMMS design are free to use them as required.

2. DESCRIPTION OF THE STANDARD ORGANIZATION

The format for specifying an abstract interface consists of the following sections:

2.1 Introduction

This section introduces, in informal prose, the features provided by the module. It may define basic concepts that are used in the rest of the specification.

2.2 Interface Overview

The interface overview section includes tables that provide an overview of facilities provided by the module. Readers familiar with the module interface can use these tables to refresh their memories about particular facts without having to reread the longer explanations in later sections. The interface overview may contain any of the following subsections:

Access Program Table

Table 2 shows the form for the access program table. This table lists all access programs provided by the module, as well as the number, data type, and semantics of the parameters. Access programs can change or retrieve information that is stored in a module's internal storage. Access program names begin and end with brackets that show when they can be used: ++ brackets indicate programs that may be invoked only at system-generation time; + brackets enclose programs that are executable at run time. The access program table contains an entry for each access program provided by the module; each entry includes the program name, parameter data, and exceptions (discussed more fully later) associated with the program.

There are three types of access programs. Each type is characterized by the facilities offered to user programs, the effects on other access programs provided by the module, the information required to specify them, and the naming conventions.

Value Programs — These programs deliver values to user programs via output parameters. A call to a value program has no effect on subsequent calls to that program or on any other program of the same module. Semantics of value programs are given in the dictionary definition(s) of the term(s) used to denote the output parameter(s). Value program names usually begin with "Get" for Get value.

Effect Programs — These programs enable user programs to affect the future operation of the module by passing it information or giving it commands. Effect programs may affect the values returned by subsequent calls to value programs, may change the values shown by display devices, or may affect the current operating state of the module. These programs do not return values themselves. The description column in the access program table can be left blank for these programs whenever the program effects section adequately defines the parameter meanings. The names of effect programs usually begin with "Set" for Set value.

Hybrid Programs — These programs have characteristics of both value and effect programs: they return values and affect the future operation of the module. These programs will usually be described both by parameter-information entries in the access program table and descriptions in the effects section.

The "Exceptions" column lists all the Undesired Events, User Exceptions, and Security Exceptions that can occur during execution of the specified access program.

Parameters

Although the access program table specifies the required data type for each parameter, there may be further requirements on the values provided as input parameters. These legality conditions (e.g., asserting that a particular parameter must be positive) are given in this section. If no restrictions on any input parameter's value is necessary, this section is omitted.

Effects

This section specifies the effects (semantics) of invoking a hybrid or effect access program. The effects are specified completely in terms of changes or results that are completely observable by using software or by a human observer. It is basic to the information-hiding methodology that no information about the implementation or other hidden aspects of a module be divulged in this section. Effects may be given by specifying changes in the values that will subsequently be returned by access programs, or events that will occur at a later time. An example of a human-observable effect is the positioning of a symbol on a display. If any run-time undesired events are enabled or disabled as a result of invoking the program, that is also described here. If there are no effect or hybrid access programs in the table, this section is omitted.

The access program table may be split into subtables of access programs that have enough in common that it is useful to study them together. In this case, the effects of the access programs immediately follow each subtable.

2.3 Local Data Types

For every program parameter, a type is specified in the interface overview. This section of the specification defines the data types that are used in communicating with the module. All such data types are described in this section except those that are defined in another module interface specification, in which case a reference to that specification is to be given.

2.4 Dictionary

This section of the specification defines terms that appear using the !+term+! and !!term!! notation in other sections of the specification.

An item of the form !+term+! is used in the access program table to name an output parameter of a program. The dictionary definition of such a !+term+!, then, defines the value returned by the access

Program	Parameters	Description	Exceptions
++*program1*++ or +*program1*+	**p1:***type1*;*K*	*info1*	%%*name1*%% or %*name1*% or
	p2:*type2*;*K*	*info2*	%(*name1*)% or %{*name1*}%
	.	.	. . .
	.	.	%%*nameM*%% or %*nameM*% or
	p*N1***:***typeN1*;*K*	*infoN1*	%(*nameM*)% or %{*nameM*}%
++*program2*++ or +*program2*+	**p1:***type1*;*K*	*info1*	
	p2:*type2*;*K*	*info2*	
	.	.	
	.	.	
	p*N2***:***typeN2*;*K*	*infoN2*	
.			
.			
.			
++*programG*++ or +*programG*+	**p1:***type1*;*K*	*info1*	
	p2:*type2*;*K*	*info2*	
	.	.	
	.	.	
	p*NJ***:***typeNG*;*K*	*infoNG*	

Effects

++*programI*++ or +*programI*+	*formal or informal description of externally observable results of invoking program* $I (1 \leq I \leq G)$

Table 2 — Access program table format

Legend for Table 2

Bold-faced symbols are **required**. Other names and letters are defined as follows:

G	number of programs in the group, where group is defined as a set of programs with the same entries in the exceptions column; different groups are separated by a horizontal line in the table.
programJ	name of the Jth program in the group, where J = 1,...,G. If the name contains ++ brackets, that program may only be invoked at system-generation time; that is, that program will exist only in the support software prior to the time the software is loaded onto the target machine. A name with + brackets may be invoked at run-time; that is, that program will be available for invocation on the target machine.
NJ	number of parameters for the Jth program. If zero, the parameter column is empty for the program.
pL	the Lth parameter of a program, where L = 1,...,NJ
typeL	type of parameter pL: the name of a data type provided either by this module or another. If provided by this module, it will be defined in the Local Data Types section of the specification. If not provided by the module, it will be defined in the module corresponding to the module prefix provided in its name.
K	I, O, IO for input, output, and input-output parameter. Programs receive the values of input parameters and deliver the values of output parameters. Input-output parameters serve both purposes.
infoL	definition of the meaning of parameter pL; may be an entry in the specification's dictionary (!+entry L+!) or an expression involving other parameters, such as p1 + p2; infoL may be omitted for any parameter whose meaning is given in the effects section, or it may be an informal description summarizing the program effect description.
M	number of exception dictionary entries defined for the group
nameE	Entry E in the specification's exception dictionary, where E = 1,2,...M; the dictionary entry defines the circumstances that cause a program call to be an exception. If the name contains %% brackets, the exception will be detected before run-time, and the user may not provide a run-time program to handle the exception. If the name has % brackets (with or without the parentheses or braces), the exception may not be detected until run-time, and the user is obligated to provide a run-time exception-handling program for it. Naturally, system-generation-time programs can only have system-generation-time UEs associated with them. There are three classes of exceptions: Undesired Events, Security Exceptions, and User Exceptions (See sections 2.5, 2.6, and 2.7 for more details). Undesired Events are denoted by % or %% brackets alone. Security Exceptions and User Exceptions are surrounded by %(and)%.

program via the output parameter. This gives the semantics of the program. As in program effects, the definition is given only in terms that can be tested by the software or a human user.

A !!term!! may be used anywhere in the specification (except to describe an output parameter of a program) to take the place of a specialized technical definition that would otherwise have to be repeated.

The definitions may be in either prose or a formal notation, given in alphabetical order by term.

2.5 Undesired Event Dictionary

An undesired event (UE) is an exception resulting from a hardware or software error that, once detected, may be handled by a "trap" that will effect recovery from the error. For a more complete discussion of UEs see [UE].

A UE occurs when an assumption about an undesired event is violated, usually when an access program is called with an incorrect parameter or in a state in which it cannot be executed successfully. This section defines the conditions that correspond to each undesired event reported by the module.

A UE is considered **enabled** when the UE may occur and **inhibited** when it is impossible for it to occur. Some UEs are always enabled. Some UEs are inhibited or enabled by access programs (user-controlled state UEs). Some UEs are inhibited or enabled by changes detected within the module (internal state UEs), and their status is available via access programs. If some assurance can be given that a UE will never occur, it is permissible to remove its detection code from a production version of the system to improve performance.

This section defines the %term% or %%term%% entries in the access program table by stating the violation that each one represents. A UE of the form %%term%% will be detected at system generation time. A UE of the form %term% may not be detected until run time. The specification describes user-controlled state UEs in terms of the commands that inhibit or enable them and internal state UEs in terms of the value programs that reveal whether the UE is currently inhibited or enabled.

2.6 User Exception Dictionary

A user exception occurs when a condition motivated by functional requirements (outside the security model constraints) is violated. It results from an action on a human user's part rather than an error in the calling program. Programs that use the module may depend on it for detection and reporting of user exceptions. This section defines the conditions that correspond to each user exception reported by the module. The access program is required to detect and report each user exception when it occurs; therefore, a user exception is always **enabled**.

This section defines the %(term)% entries in the access program table by stating the violation that each one represents. A user exception can not be detected until run time.

2.7 Security Exception Dictionary

A security exception occurs when a condition motivated by security model constraints is violated. It results from an action on a human user's part rather than an error in the calling program. Programs that use the module may depend on it for the detection and reporting of such exceptions. This section defines the conditions that correspond to each security exception reported by the module. The access program is required to detect and report each security exception when it occurs; therefore, a security exception is always **enabled**.

This section defines the %{term}% entries in the access program table by stating the violation that each one represents. A security exception can not be detected until run time.

2.8 System Generation Parameters

This section describes those externally visible characteristics of the module that can be changed by assigning values to parameters at system generation time. Each parameter is named, its data type is given, and its meaning is described. These parameters are denoted by #term#, and may be used as symbolic constants by users of the module.

2.9 Interface Design Issues

This prose section describes any alternative designs that were considered and records the reason for their rejection. The section serves as a history of design decisions, so that issues are not considered repeatedly. It serves as a design rationale providing guidance to maintenance programmers revising the program.

2.10 Implementation Notes

This prose section contains implementation notes. During the design of the module interface, certain facts or ideas may come to the designer's attention, ideas that would be necessary or useful to future implementers, and these are noted in this section.

2.11 Assumptions Lists

The information in the assumptions lists may be redundant. It is implied by the description of the facilities specified in the rest of the section. The purpose of the assumption list is to serve as an explicit medium for review by nonprogrammers. Supplying this list for the benefit of nonprogrammers necessitates the redundancy.

This section comprises four prose subsections.

Basic Assumptions

These assumptions contain information that users of the module may assume will never change. In the case of hardware-hiding modules [MG], it consists of information that will remain true about the interface even if the hidden hardware is replaced or modified. In the case of requirements-hiding modules, it consists of information that will remain true even if the hidden requirements are changed. In the case of software decision-hiding modules, it consists of information that will remain true even if the hidden software decisions are changed.

The assumptions relate to the normal use and operation of the module. A basic assumption will fall into one of two categories: implementability (an assumption that the module's facilities can be implemented efficiently), and sufficiency (an assumption that the given facilities are all the user will ever need). Specifically, they may concern: (a) information available from the module; (b) information that must be supplied to the module; (c) tasks that can be performed by the module; (d) operating states of the module and how they affect the information available and the information required; or (e) failure states of the module and how they affect the information available.

Assumptions About Undesired Events

This section lists assumptions describing **incorrect** usage of the module at run-time. Violation of each assumption is associated with a run-time undesired event. The development version of the system will be designed report the undesired event whenever a violation occurs. In the production version of the system, the undesired event-handling code will be removed, and violations of the assumptions in this section will result in unpredictable behavior.

Assumptions About User Exceptions

This section lists assumptions describing the functional usage of the module at run-time (that are not security-related). The system will be designed to report the user exception whenever a violation occurs.

Assumptions About Security Exceptions

This section lists assumptions describing the func·ional usage of the module at run-time that are security-related. The system will be designed to report the security exception whenever a violation occurs.

2.12 Unimplemented Features

It is often the case that a particular version of the system may not need features of a module that are likely to be needed in other versions. The interface descriptions specify all the features of a module, but in this section those features that will not be available in the initial version are identified. The programmer can use this information about likely future additions to design his software for easier extension.

For each unimplemented feature, an undesired event is specified (in the access program table) that will be raised if a programmer attempts to use the feature.

2.13 Efficiency Guide

This section contains specifications of the run-time resources (memory, cpu time) consumed by invoking the access programs on the module's interface. If precise measures are not available, then guidelines for more efficient usage will be given. This, of course, may be implementation dependent.

2.14 Mapping to Requirements

This section contains a mapping between the interface programs and the system requirements [REQ] to which they can be traced. In instances where the interface programs do not map directly to the system requirements, a [illegible] statement of how the interface programs map indirectly to the requirements must be given.

2.15 Faciliti[illegible] [illegible]dex

After all the submodules in the document have been specified using the foregoing scheme, an index is provided that shows where in the document a particular name is defined. The index includes a list of access programs, local data types, dictionary items, undesired event and exception names, and system generation parameters. The system generation parameter list includes a range of expected values for each parameter.

2.16 References

Any other documents to which a reference is made shall be defined here.

3. NOTATION CONVENTIONS

Table 3 lists the notational brackets used and indicates what section(s) of an interface specification gives relevant information.

Notation	Meaning	Where to Look It Up
++name++	A module access program that may only be invoked at system generation time	Section 2
+name+	A module access program that may be invoked at run-time	Section 2
+GetName+ or ++GetName++	A value access program; does not change the state of the module, but returns a value described in the dictionary	Sections 2.5
+SetName+ or ++SetName++	An effect access program; changes the module state as described in Section 2 Usually returns no value.	Section 2
%%name%%	An undesired event that will be detected at system-generation time	Sections 2.5
%name%	An undesired event that may not be detected until run-time	Sections 2.5
%(name)%	A user exception	Sections 2.6
%{name}%	A security exception	Sections 2.7
!+name+!	The name of a value produced by a module's access program; its definition is given in the specification's dictionary section.	Sections 2.4
!!name!!	Used to denote a term with a specialized definition that appears frequently in the specification; its definition is given in the specification's dictionary section.	Section 4
#name#	The name of a system-generation time parameter	Section 6

Table 3 — Notation Convention Table

The names of all access programs, local types, dictionary terms, exceptions, and system generation parameters referenced from other modules shall be prefixed by the identifier of the second-level module in which the name is defined, followed by a period. This will result in names of the form **X**.name, which shall be defined in the relevant section of reference [**X**]. For example, the definition of !!SP.classification!! can be found by looking it up in the Security Policy Interface, [SP].

4. EXAMPLE

The following is an example to illustrate the form of the first eight sections of a specification of an abstract interface. (The remaining sections of the specification, composed of an index and prose paragraphs, are not shown.) It has been taken from Chapter Two of the "Interface Specifications for the Entity Monitor Module". The interface specification [SP] will provide additional examples.

User Primitives Module

1.1.1. Introduction

This module provides facilities to authorize new users and associate them them with userID's, !!clearance!!s and specific rights.

1.1.2. Interface Overview

1.1.2.1. ACCESS PROGRAM TABLE - User Functions

Program	*Parameters*	*Description*	*Exceptions*
+CreateUser+	p1: ABM.name; I	!!user name!!	%(user name taken)% %{user not sso}%
+DestroyUser+	p1: userID; I	!!user identifier!!	%(no such user p1)% %{user not sso}%
+ExistsUser+	p1: userID; I p2: ABM.boolean; O	!!user identifier!! !+user exists+!	%(no such user p1)%
+ExistsUserName+	p1: ABM.name; I p2: ABM.boolean; O	potential !!user name!! !+user name exists+!	none
+GetCurrentUser+	p1: userID; O	!+current user+!	none
+GetUserID+	p1: ABM.name; I p2: userID; O	!!user name!! !!user identifier!!	%(no such user name p1)%
+GetUserName+	p1: userID; I p2: ABM.name; O	!!user identifier!! !!user name!!	%(no such user p1)%
+ComparePassword+	p1: userID; I p2: ABM.string; I	!!user identifier!! !!password!!	

	p3: ABM.boolean; O	!+user authenticated+!	%c(no such user p1)%c %c{not the current user or sso}%c
+SetPassword+	p1: userID: I p2: ABM.string; I	!!user identifier!! !!password!!	%c(no such user p1)%c %c{not the current user or sso}%c
+GetUserClearance+	p1: userID: I p2: ABM.securityLev; O	!!user identifier!! !+user clearance+!	%c(no such user p1)%c
+SetUserClearance+	p1: userID; I p2: ABM.securityLev; I	!!user identifier!! !!clearance!!	%c(no such user p1)%c %c{user not sso}%c
+GetCurrentTerminal+	p1: ABM.cardinal; I	!!ICL parameter position!!	%cnot logged in%c
+GetDefaultPrinter+	p1: ABM.cardinal; I	!!ICL parameter position!!	%c(printer not set)%c
+SetDefaultPrinter+	p1: ent; I	!!default printer!!	%cnot a device%c %c(no such entity p1)%c

——————— *Effects* ———————

+CreateUser+	+ExistsUserName+(p1)' = true
	+ExistsUser+(+GetUserID+(p1))' = true
	+GetUserClearance+(+GetUserID+(p1))' = $SystemLow$
	r . +InAuthRole+(+GetUserID+(p1), r)' = false
	r.+InRole+(+GetUserID+(p1), r)' = false
	(+GetUserName+(+GetUserID+(p1)))' = p1
+DestroyUser+	+ExistsUser+(p1)' = false
	+ExistsUserName+(+GetUserName+(p1)[s])[s'] = false
+SetPassword+	+ComparePassword+(p1,p2)' = true
	i. i≠p2. +ComparePassword+(p1,i)' = false
+SetUserClearance+	+GetUserClearance+(p1)' = p2

+GetCurrentTerminal+	+ArgEnt+(p1)' = !!current terminal!!
+GetDefaultPrinter+	+ArgEnt+(p1)' = !!default printer!!
+SetDefaultPrinter+	+GetDefaultPrinter+(+CurrentUser+(s))(s') = p1

1.1.3. Local type definitions

userID	The internal representation of a !!user identifier!!.

1.1.4. Dictionary

!!clearance!!	A !!security level!! denoting the level of trust associated with a user.
!!correct password!!	A !!password!! associated with a !!user identifier!!. When the system is initialised the resulting system state determines what !!user identifier!!s and !!password!!s are considered correct for what user identifiers. After initialization the correct password is the password most recently associated with that user by +SetPassword+ or the initial setting if no subsequent calls to +SetPassword+ have been made.
!!current terminal!!	The handle to the !!device!! that the !!current user!! is logged in on. Any user actions sensed by this !!device!! are assumed to be originated by the !!current user!!.
!!current user!!	*The user on whose behalf the client program making the call is running.*
!+current user+!	Holds a !!user identifier!! indicating the !!current user!!.
!!default printer!!	A !!device!! that has been set aside for the !!current user!! to use as a printer.
!!password!!	A string used to authenticate the user. Presentation of a !!correct password!! for a particular !!user identifier!! is considered sufficient evidence that the presenter is the person denoted by that identifier.
!+user authenticated+!	True if the !!password!! presented is the !!correct password!! for the !!user identifier!!. False if no password is associated with the given !!user identifier!! or the passwords presented is not the !!correct password!!.
!+user clearance+!	The !!clearance!! stored in the system for the given !!user identifier!!.
!+user exists+!	True if and only if the given !!user identifier!! has been successfully created by a +CreateUser+ command and has not been destroyed by a subsequent +DestroyUser+ command.
!!user identifier!!	An internally generated token that uniquely identifies a human user on the system.
!!user name!!	A string that uniquely identifies a human user of the system.
!+user name exists+!	True if and only if the given !!user name!! corresponds to an existing !!user identifier!!.

1.1.5. Undesired Event dictionary

%not logged in% The !!current user!! is undefined; i.e., not logged in.

1.1.6. User Exception dictionary

%(no such user p1)%
%(no such user p2)% The identifier given for the !!user identifier!! does not refer to any defined user.

NOT +ExistsUser+(x)

where "x" is replaced by "p1" or "p2".

%(no such user name p1)%
The string given as a !!user name!! is unknown to the system.

NOT +ExistsUserName+(p1)

%(printer not set)% No value for the !!default printer!! has been previously set.

%(user name taken)% The given !!user name!! is already associated with a defined !!user identifier!!.

+ExistsUser+(+GetUserID+(p1))

1.1.7. Security Exception dictionary

%{not the current user or sso}%
An attempt was made to perform an operation that may only be done by the 'System Security Officer' or for the userID that matches that of the !!current user!!.

NOT (+GetCurrentUser+ = p1) AND NOT (+InRole+(+GetCurrentUser+,"sso"))

%{user not sso}% The !!current user!! is attempting to perform an operation that may only be done by the 'System Security Officer', and it is not among his !!active roles!!.

1.1.8. System Generation Parameters

none

5. ACKNOWLEDGEMENTS

We owe thanks to Paul C. Clements, R. Alan Parker, David L. Parnas, John Shore, and Kathryn H. Britton who are the authors of the original paper [SCRSO] on which this paper is based. Much of their original prose is incorporated in this document. They are, of course, not responsible for any errors that our revisions and additions may have introduced into the structure they developed.

REFERENCES

[SCRSO]	P.C. Clements, R. A. Parker, D. L. Parnas, J. Shore, K. H. Britton, "A Standard Organization for Specifying Abstract Interfaces," NRL Report 8815, June 14, 1984 (Release 2.11).
[CRIT]	D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, **15** (12), 1053-1058 (1972).
[SAP]	M.R. Cornwell, "Security Architecture Proof for the SMMS Full Scale Prototype," NRL Technical Memorandum 5590-94, March 17 1988.
[MG]	M.R. Cornwell, A.L. Evans, "SMMS Software Module Guide," NRL Memorandum Report to appear.
[SP]	M.R. Cornwell, A.L. Evans, "Interface Specifications for the Entity Monitor Module," NRL Memorandum Report to appear.
[UE]	D.L. Parnas, H. Wuerges, "Response to Undesired Events in Software Systems," *Proc. Second Int. Conf. Software Eng.*, pp. 437-446, 1976.
[REQ]	J.T.Quinn, M.R.Cornwell, C.L.Heitmeyer, C.E. Landwehr J.D. McLean, " Software Requirements for the Secure Military Message System Family, " NRL Memorandum Report to appear.
[ABS]	K. H. Britton, R. A. Parker, and D. L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules, " *Proc. Fifth Int. Conf. Software Engineering* pp. 195-206, 1981.